Planning by Prioritized Sweeping with Small Backups

Harm van Seijen   HARM.VANSEIJEN@UALBERTA.CA
Richard S. Sutton   SUTTON@CS.UALBERTA.CA
Department of Computing Science, University of Alberta, Edmonton, Alberta, T6G 2E8, Canada

Abstract

Efficient planning plays a crucial role in model-based reinforcement learning. Traditionally, the main planning operation is a full backup based on the current estimates of the successor states. Consequently, its computation time is proportional to the number of successor states. In this paper, we introduce a new planning backup that uses only the current value of a single successor state and has a computation time independent of the number of successor states. This new backup, which we call a *small backup*, opens the door to a new class of model-based reinforcement learning methods that exhibit much finer control over their planning process than traditional methods. We empirically demonstrate that this increased flexibility allows for more efficient planning by showing that an implementation of prioritized sweeping based on small backups achieves a substantial performance improvement over classical implementations.

1. Introduction

In *reinforcement learning* (RL) (Kaelbling et al., 1996; Sutton & Barto, 1998), an agent seeks an optimal control policy for a sequential decision problem in an initially unknown environment. The environment provides feedback on the agent's behavior in the form of a reward signal. The agent's goal is to maximize the expected *return*, which is the discounted sum of rewards over future timesteps. An important performance measure in RL is the *sample efficiency*, which refers to the number of environment interactions that is required to obtain a good policy.

Many solution strategies improve the policy by iteratively improving a *state-value* or *action-value function*, which provide estimates of the expected return under a given policy for (environment) states or state-action

pairs, respectively. Different approaches for updating these value functions exist. In terms of sample efficiency, one of the most effective approaches is to estimate the environment model using observed samples and to compute, at each time step, the (action-)value function that is optimal with respect to the model estimate using planning techniques. A popular planning technique used for this is *value iteration* (VI) (Sutton, 1988; Watkins, 1989), which performs sweeps of backups through the state or state-action space, until the (action-)value function has converged.

A drawback of using VI is that it is computationally very expensive, making it infeasible for many practical applications. Fortunately, efficient approximations can be obtained by limiting the number of backups that is performed per timestep. A very effective approximation strategy is *prioritized sweeping* (Moore & Atkeson, 1993; Peng & Williams, 1993), which prioritizes backups that are expected to cause large value changes. This paper introduces a new backup that enables a dramatic improvement in the efficiency of prioritized sweeping.

The main idea behind this new backup is as following. Consider that we are interested in some estimate A that is constructed from a sum of other estimates X_i. The estimate A can be computed using a *full backup*:

$$A \leftarrow \sum_i X_i \,.$$

If the estimates X_i are updated, A can be recomputed by redoing the above backup. Alternatively, if we know that only X_j received a significant value change, we might want to update A for only X_j. Let us indicate the old value of X_j, used to construct the current value of A, as x_j. A can then be updated by subtracting this old value and adding the new value:

$$A \leftarrow A - x_j + X_j \,.$$

This kind of backup, which we call a *small backup*, is computationally cheaper than the full backup. The

trade-off is that, in general, more memory is required for storing the estimates x_i associated with A. In planning, where the X estimates correspond to state-value estimates and A corresponds to a state or state-action estimate, this is not a serious restriction, because a full model is stored already. The additional memory required has the same order of complexity as the memory required for storage of the model.

The core advantage of small backups over full backups is that they enable finer control over the planning process. This allows for more effective update strategies, resulting in improved trade-offs between computation time and quality of approximation of the VI solution (and hence sample efficiency). We demonstrate this empirically by showing that a prioritized sweeping implementation based on small backups yields a substantial performance improvement over the two classical implementations (Moore & Atkeson, 1993; Peng & Williams, 1993).

In addition, we demonstrate the relevance of small backups in domains with severe constraints on computation time, by showing that a method that performs one small backup per time step has an equal computation time complexity as TD(0), the classical method that performs one *sample backup* per timestep. Since sample backups introduce sampling variance, they require a step-size parameter to be tuned for optimal performance. Small backups, on the other hand, do not introduce sampling variance, allowing for a parameter-free implementation. We empirically demonstrate that the performance of a method that performs one small backup per time step is similar to the optimal performance of TD(0), achieved by carefully tuning the step-size parameter.

2. Reinforcement Learning Framework

RL problems are often formalized as *Markov decision processes* (MDPs), which can be described as tuples $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ consisting of $\mathcal{S}$, the set of all states; $\mathcal{A}$, the set of all actions; $\mathcal{P}_{sa}^{s'} = Pr(s'|s,a)$, the transition probability from state $s \in \mathcal{S}$ to state s' when action $a \in \mathcal{A}$ is taken; $\mathcal{R}_{sa} = E\{r|s,a\}$, the reward function giving the expected reward r when action a is taken in state s; and γ, the discount factor controlling the weight of future rewards versus that of the immediate reward.

Actions are selected at discrete timesteps $t = 0, 1, 2, ...$ according to a *policy* $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, which defines for each action the selection probability conditioned on the state. In general, the goal of RL is to improve the policy in order to increase the *return G*, which is

the discounted cumulative reward

$$G_t = r_{t+1} + \gamma\, r_{t+2} + \gamma^2\, r_{t+3} + ... = \sum_{k=1}^{\infty} \gamma^{k-1}\, r_{t+k}\,,$$

where r_{t+1} is the reward received after taking action a_t in state s_t at timestep t.

The *prediction task* consists of determining the *value function $V^\pi(s)$*, which gives the expected return when policy π is followed, starting from state s. $V^\pi(s)$ can be found by making use of the *Bellman equations* for state values, which state the following:

$$V^\pi(s) = \mathcal{R}_s + \gamma \sum_{s'} \mathcal{P}_s^{s'} V^\pi(s')\,, \qquad (1)$$

where $\mathcal{R}_s = \sum_a \pi(s,a)\mathcal{R}_{sa}$ and $\mathcal{P}_s^{s'} = \sum_a \pi(s,a)\mathcal{P}_{sa}^{s'}$.

Model-based methods use samples to update estimates of the transition probabilities, $\hat{\mathcal{P}}_s^{s'}$, and reward function, $\hat{\mathcal{R}}_s$. With these estimates, they can iteratively improve an estimate V of V^π, by performing *full backups*, derived from Equation (1):

$$V(s) \leftarrow \hat{\mathcal{R}}_s + \gamma \sum_{s'} \hat{\mathcal{P}}_s^{s'} V(s')\,. \qquad (2)$$

In the *control task*, methods often aim to find the optimal policy π^*, which maximizes the expected return. This policy is the greedy policy with respect to the optimal *action-value function $Q^*(s,a)$*, which gives the expected return when taking action a in state s, and following π^* thereafter. This function is the solution to the *Bellman optimality equation* for action-values:

$$Q^*(s,a) = \mathcal{R}_{sa} + \gamma \sum_{s'} \mathcal{P}_{sa}^{s'} \max_{a'} Q^*(s',a')\,. \qquad (3)$$

The optimal value function is related to the optimal action-value function through: $V^*(s) = \max_a Q^*(s,a)$.

Model-based methods can iteratively improve estimates Q of Q^* by performing full backups derived from Equation (3):

$$Q(s,a) \leftarrow \hat{\mathcal{R}}_{sa} + \gamma \sum_{s'} \hat{\mathcal{P}}_{sa}^{s'} \max_{a'} Q(s',a')\,, \qquad (4)$$

where $\hat{\mathcal{R}}_{sa}$ and $\hat{\mathcal{P}}_{sa}^{s'}$ are estimates of $\mathcal{R}_{sa}$ and $\mathcal{P}_{sa}^{s'}$, respectively.

Model-free methods do not maintain an model estimate, but update a value function directly from samples. A classical example of a *sample backup*, based on sample (s, r, s') is the TD(0) backup:

$$V(s) \leftarrow V(s) + \alpha\,(r + \gamma V(s') - V(s))\,, \qquad (5)$$

where α is the step-size parameter.

3. Small Backup

This section introduces the small backup. We start with small state-value backups for the prediction task. Section 3.3 discusses small action-value backups for the control task.

3.1. Value Backups

In this section, we introduce a small backup version of the full backup for prediction (backup 2). In the introduction, we showed that a small backup requires storage of the component values that make up the current value of a variable. In the case of a small value backup, the component values correspond to the values of successor states. We indicate these values by the function $U_s : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$. So, $U_s(s')$ is the value estimate of s' associated with s.

Using U_s, $V(s)$ can be updated with only the current value of a single successor state, s', as demonstrated by the following theorem. The three backups shown in the theorem form together the small backup.

Theorem 3.1 *If the current relation between $V(s)$ and U_s is given by*

$$V(s) = \hat{\mathcal{R}}_s + \gamma \sum_{s''} \hat{\mathcal{P}}_s^{s''} U_s(s''), \qquad (6)$$

then, after performing the following backups:

$$
\begin{aligned}
tmp &\leftarrow V(s') & (7)\\
V(s) &\leftarrow V(s) + \gamma \mathcal{P}_s^{s'}[V(s') - U_s(s')] & (8)\\
U_s(s') &\leftarrow tmp, & (9)
\end{aligned}
$$

relation (6) still holds, but $U_s(s')$ is updated to $V(s')$.

Proof Backup (8) subtract the component in relation (6) corresponding to s' from $V(s)$ and adds a new component based on the current value estimate of s':

$$V(s) \leftarrow V(s) - \gamma \hat{\mathcal{P}}_s^{s'} U_s(s') + \gamma \hat{\mathcal{P}}_s^{s'} V(s').$$

Hence, relation (6) is maintained, while $U_s(s')$ is updated. Note that $V(s')$ needs to be stored in a temporary variable, since backup (8) can alter the value of $V(s')$ if $s' = s$. ∎

3.2. Value Correction after Model Update

Theorem 3.1 relies on relation (6) to hold. If the model gets updated, this relation now longer holds. In this section, we discuss how to restore relation (6) in a computation-efficient way for the commonly used model estimate:

$$
\begin{aligned}
\hat{\mathcal{P}}_s^{s'} &\leftarrow N_s^{s'}/N_s & (10)\\
\hat{\mathcal{R}}_s &\leftarrow R_s^{sum}/N_s, & (11)
\end{aligned}
$$

where N_s counts the number of times state s is visited, $N_s^{s'}$ counts the number of times s' is observed as successor state of s, and R_s^{sum} is the sum of observed rewards for s.

Theorem 3.2 *If currently, the following relation holds:*

$$V(s) = \hat{\mathcal{R}}_s + \gamma \sum_{s''} \hat{\mathcal{P}}_s^{s''} U_s(s''),$$

and a sample (s, r, s') is observed, then, after performing the backups:

$$N_s \leftarrow N_s + 1; \quad N_s^{s'} \leftarrow N_s^{s'} + 1 \qquad (12)$$

$$V(s) \leftarrow \Big[V(s)(N_s - 1) + r + \gamma U_s(s')\Big]/N_s. \quad (13)$$

the relation still holds, but with updated values for $\hat{\mathcal{R}}_s$ and $\hat{\mathcal{P}}_s^{s''}$.

Proof (sketch) Backup (13) updates $V(s)$ by computing a weighted average of $V(s)$ and $r + \gamma U_s(s')$. The value change this causes is the same as the value change caused by updating the model and then performing a full backup of s based on U_s. ∎

Algorithm 1 shows pseudo-code for a general class of prediction methods based on small backups. Surprisingly, while it is a planning method, $\hat{\mathcal{R}}_s$ is never explicitly computed, saving time and memory. Note that the computation per time step is fully independent of the number of successor states. Members of this class need to specify the number of iterations (line 8) as well as a strategy for selecting state-successor pairs (line 9).

Algorithm 1 Prediction with Small Backups

1: initialize $V(s)$ arbitrarily for all s
2: initialize $U_s(s') = V(s')$ for all s, s'
3: initialize $N_s, N_s^{s'}$ to 0 for all s, s'
4: **loop** {over timesteps}
5:     observe transition (s, r, s')
6:     $N_s \leftarrow N_s + 1; \quad N_s^{s'} \leftarrow N_s^{s'} + 1$
7:     $V(s) \leftarrow \Big[V(s)(N_s - 1) + r + \gamma U_s(s')\Big]/N_s$
8:     **loop** {for a number of iterations}
9:       select a pair $(\bar{s}, \bar{s}')$ with $N_{\bar{s}}^{\bar{s}'} > 0$
10:      $tmp \leftarrow V(\bar{s}')$
11:      $V(\bar{s}) \leftarrow V(\bar{s}) + \gamma N_{\bar{s}}^{\bar{s}'}/N_{\bar{s}} \cdot [V(\bar{s}') - U_{\bar{s}}(\bar{s}')]$
12:      $U_{\bar{s}}(\bar{s}') \leftarrow tmp$
13:     **end loop**
14: **end loop**

3.3. Action-value Backups

Before we can discuss small action-value backups, we have to discuss a more efficient implementation of the

full action-value backup. Backup (4) has a computation time complexity of $\mathcal{O}(|\mathcal{S}||\mathcal{A}|)$. A more efficient implementation can be obtained by storing state-values, besides action-values, according to $V(s) = \max_a Q(s, a)$. Backup (4) can then be implemented by:

$$Q(s, a) \quad \leftarrow \quad \hat{\mathcal{R}}_{sa} + \gamma \sum_{s'} V(s') \qquad (14)$$

$$V(s) \quad \leftarrow \quad \max_{a'} Q(s, a) \,. \qquad (15)$$

The combined computation time of these backups is $\mathcal{O}(|\mathcal{S}| + |\mathcal{A}|)$, a considerable reduction.

Backup (14) is similar in form as the prediction backup. Hence, we can make a small backup version of it similar to the one in the prediction case. The theorems below are the control versions of the theorems for the prediction case. They can be proven in a similar way as the prediction theorems.

Theorem 3.3 *If the current relation between $Q(s, a)$ and U_{sa} is given by*

$$Q(s, a) \leftarrow \hat{\mathcal{R}}_{sa} + \gamma \mathcal{P}_{sa}^{s''} \sum_{s''} U_{sa}(s'') \,, \qquad (16)$$

then, performing the following backups:

$$Q(s, a) \quad \leftarrow \quad Q(s, a) + \gamma \mathcal{P}_{sa}^{s'}[V(s') - U_{sa}(s')]$$
$$U_{sa}(s') \quad \leftarrow \quad V(s') \,,$$

maintains this relation while updating $U_{sa}(s')$ to $V(s')$.

Theorem 3.4 *If relation (16) holds and a sample (s, a, r, s') is observed, then, after performing backups*

$$N_{sa} \quad \leftarrow \quad N_{sa} + 1; \quad N_{sa}^{s'} \leftarrow N_{sa}^{s'} + 1$$
$$Q(s, a) \quad \leftarrow \quad \left[Q(s, a)(N_{sa} - 1) + r + \gamma U_{sa}(s') \right]/N_{sa} \,,$$

relation (16) still holds, but with updated values for $\hat{\mathcal{R}}_{sa}$ and $\hat{\mathcal{P}}_{sa}^{s''}$.

A small action-value backup is a finer-grained version of backup (14): performing a small backup of $Q(s, a)$ for each successor state is equivalent (in computation time complexity and effect) as performing backup (14) once. While in principle, backup (15) can be performed after each small backup, it is not very efficient to do so, since small backups make many small changes. More efficient planning can be obtained when backup (15) is performed only once in a while.

In Section 4, we discuss an implementation of prioritized sweeping based on small action-value backups.

3.4. Small Backups versus Sample Backups

A small backup has in common with a sample backup that both update a state value based on the current value of only one of the successor states. In addition, they share the same computation time complexity and their effect is in general smaller than that of a full backup.

A disadvantage of a sample backup, with respect to a small backup, is that it introduces sampling variance, caused by a stochastic environment. This requires the use of a step-size parameter to enable averaging over successor states (and rewards). A small backup does not introduce sampling variance, since it is implicitly based on an expectation over successor states. Hence, it does not require tuning of a step-size parameter for optimal performance.

A second disadvantage of a sample backup is that it affects the perceived distribution over action outcomes, which places some restrictions on reusing samples. For example, a model-free technique like experience replay (Lin, 1992), which stores experience samples in order to replay them at a later time, can introduce bias, which reduces performance, if some samples are replayed more often than others. For small backups this does not hold, since the process of learning the model is independent from the backups based on the model. This allows small backups to be combined with effective selection strategies like prioritized sweeping.

4. Prioritized Sweeping with Small Backups

Prioritized sweeping (PS) makes the planning step of model-based RL more efficient by using a heuristic (a 'priority') for selecting backups that favours backups that are expected to cause a large value change. A priority queue is maintained that determines which values are next in line for receiving backups.

There are two main implementations: one by Moore & Atkeson (1993) and one by Peng & Williams (1993) [1]. All PS methods have in common that they perform backups in what we call *update cycles*. By adjusting the number of update cycles that is performed per time step, the computation time per time step can be controlled. Below, we discuss in detail what occurs in an update cycle for the two classical PS implementations.

---

[1] We refer to the version of 'queue-Dyna' for stochastic domains, which is different from the version for deterministic domains.

4.1. Classical Prioritized Sweeping Implementations

In the Moore & Atkeson implementation the elements in the queue are states and the backups are full value backups. In control, a full value backup is different from backup (2). Instead, it is equivalent (in effect and computation time) to performing backup (14) for each action, followed by backup (15). Hence, the associated computation time has complexity $\mathcal{O}(|\mathcal{S}||\mathcal{A}| + |\mathcal{A}|)$.

An update cycle consists of the following steps. First, the top state is removed from the queue, and receives a full value backup. Let s bet the top state and ΔV_s the value change caused by the backup. Then, for all predecessor state-action pairs $(\bar{s}, \bar{a})$ a priority p is computed, using:

$$p \leftarrow \mathcal{P}^s_{\bar{s}\bar{a}} \cdot |\Delta V_s| . \tag{17}$$

If $\bar{s}$ is not yet on the queue, then it is added with priority p. If $\bar{s}$ is on the queue already, but its current priority is smaller than p, then the priority of $\bar{s}$ is upgraded to p.

The Peng & Williams implementation differs from the Moore & Atkeson implementation in that the backup is not a full value backup. Instead, it is a backup with the same effect as a small action-value backup, but with a computational complexity of $\mathcal{O}(|\mathcal{S}| + |\mathcal{A}|)$. So, it is a cheaper backup than a full backup, but its value change is (much) smaller. The backup requires a state-action-successor triple. Hence, these triples are the elements on the queue. Predecessors are added to the queue with a priorities that estimate the action-value change.

4.2. Small Backup implementation

A natural small backup implementation might appear to be an implementation similar to that of Peng & Williams, but with the main backup implemented more efficiently. The low computational cost of a small backup, however, allows for a much more powerful implementation. The pseudo-code of this implementation is shown in Algorithm 2. Below, we discuss some key characteristics of the algorithm.

The computation time of a small backup is so low, that it is comparable to the priority computation in the classical PS implementations. Therefore, instead of computing a priority for each predecessor and performing a backup for the element with the highest priority in the next update cycle, we can perform a small backup for all predecessors. This raises the question of what to put in the priority queue and what type of backup to perform for the top element. The natural

answer is to put states in the priority queue and to perform backup (15) for the top state.

The priority associated with a state is based on the change in action-value that has occurred due to small backups, since the last value backup. This priority assures that states with a large discrepancy between the state value and action-values, receive a value backup first.

One surprising aspect of the algorithm is that it does not use the function U_{sa}, which forms an essential part of small action-value backups. The reason is that due to the specific backup strategy used by the algorithm, $U_{sa}(s')$ is equal to $V(s')$ for all state-action pairs (s, a) and all successor states s'. Hence, instead of using U_{sa}, V can be used, saving memory and simplifying the code.

Table 1 shows the computation time complexity of an update cycle for the different PS implementations. The small backup implementation is the cheapest one among the three.

	top-element backups	other
Moore & Atkeson	$\mathcal{O}(|\mathcal{S}||\mathcal{A}| + |\mathcal{A}|)$	$\mathcal{O}(P_{re})$
Peng & Williams	$\mathcal{O}(|\mathcal{S}| + |\mathcal{A}|)$	$\mathcal{O}(P_{re})$
small backups	$\mathcal{O}(|\mathcal{A}|)$	$\mathcal{O}(P_{re})$

Table 1. Computation time associated with one update cycle for the different PS implementations. P_{re} indicates the number of predecessors, state-action pairs that transition to the state whose value has just been updated.

5. Experimental Results

In this section, we evaluate the performance of a minimal version of Algorithm 1, as well as the performance of Algorithm 2.

5.1. Small backup versus Sample backup

We compare the performance of TD(0), which performs one sample backup per time step, with a version of Algorithm 1 that performs one small backup per time step. Specifically, its number of iterations (line 8) is 1, and the selected state-successor pair (line 9) is the pair corresponding to the most recent transition.

Their performance is compared on two evaluation tasks, both consisting of 10 states, laid out in a circle. State transitions only occur between neighbours. The transition probabilities for both tasks are generated by a random process. Specifically, the transition probability to a neighbour state is generated by a random number between 0 and 1 and normalized such that

Algorithm 2 Prioritized Sweeping with Small Backups

1: initialize $V(s)$ arbitrarily for all s
2: initialize $Q(s,a) = Q_{prev}(s,a) = V(s)$ for all s, a
3: initialize $N_{sa}, N_{sa}^{s'}$ to 0 for all s, a, s'
4: **loop** {over episodes}
5:     initialize s
6:     **repeat** {for each step in the episode}
7:         select action a, based on $Q(s,\cdot)$
8:         take action a, observe r and s'
9:         $N_{sa} \leftarrow N_{sa} + 1;$ $N_{sa}^{s'} \leftarrow N_{sa}^{s'} + 1$
10:        $Q(s,a) \leftarrow \big[Q(s,a)(N_{sa}-1)+r+\gamma V(s')\big]/N_{sa}$

11:        $p \leftarrow |Q(s,a) - Q_{prev}(s,a)|$
12:        if s not on queue or $p >$ current priority s, then promote $\bar{s}$ to p
13:        **for** a number of update cycles **do**
14:           remove top state $\bar{s}'$ from queue
15:           for all b: $Q_{prev}(\bar{s}',b) \leftarrow Q(\bar{s}',b)$
16:           $tmp \leftarrow V(\bar{s}')$
17:           $V(\bar{s}') \leftarrow \max_b Q(\bar{s}',b)$
18:           $\Delta V \leftarrow V(\bar{s}') - tmp$
19:           **for all** $(\bar{s}, \bar{a})$ pairs with $N_{\bar{s}\bar{a}}^{\bar{s}'} > 0$ **do**
20:              $Q(\bar{s},\bar{a}) \leftarrow Q(\bar{s},\bar{a}) + \gamma N_{\bar{s}\bar{a}}^{\bar{s}'}/N_{\bar{s}\bar{a}} \cdot \Delta V$
21:              $p \leftarrow |Q(\bar{s},\bar{a}) - Q_{prev}(\bar{s},\bar{a})|$
22:              if $\bar{s}$ not on queue or $p >$ current priority $\bar{s}$, then promote $\bar{s}$ to p
23:           **end for**
24:        **end for**
25:        $s \leftarrow s'$
26:     **until** s is terminal
27: **end loop**

the sum of the transition probabilities to the left and right neighbour is 1. The reward for counter-clockwise transitions is always +1. The reward for clockwise transitions is different for the two tasks. In the first task, a clockwise transition results in a reward of -1; in the second task, it results in a reward of +1. The discount factor γ is 0.95 and the initial state values are 0.

For TD(0), we performed experiments with a constant step-size for values between 0 and 1 with steps of 0.02. In addition, we performed experiments with a decaying, state-dependent step-size, according to

$$\alpha(s) = \frac{1}{d \cdot (N_s - 1) + 1} \quad , \qquad (18)$$

where N_s is the number of times state s has been visited, and d specifies the *decay rate*. We used values of d between 0 and 1 with steps of 0.02. Note that for $d = 0$, $\alpha(s) = 1$, and for $d = 1$, $\alpha(s) = 1/N_s$.

Each time a transition is observed and the corresponding backup is performed, the root-mean squared (RMS) error over all states is determined. The average RMS error over the first 10.000 transitions, normalized with the initial error, determines the performance. Figure 1 shows this performance, averaged over 100 runs. The standard error is negligible: the maximum standard error in the first task was 0.0057 (after normalization) and in the second task 0.0007. Note that the performance for $d = 0$ is equal to the performance for $\alpha = 1$, as it should, by definition. The normalized performance for $\alpha = 0$ is 1, since no learning occurs in this case.

These experiments demonstrate three things. First, the optimal step-size can vary a lot between different tasks. Second, selecting a sub-optimal step-size can cause large performance drops. Third, a small-backup, which is parameter-free, has a performance similar to the performance of TD(0) with optimized step-size. Since the computational complexity is the same, the small backup is a very interesting alternative to the sample backup in domains with tight constraints on the computation time, where previously only sample backups where viable. Keep in mind that a sample backup does require a model estimate, so if there are also tight constraints on the memory, a sample backup might still be the only option.

5.2. Prioritized Sweeping

We compare the performance of prioritized sweeping with small backups (Algorithm 2) with the two classical implementations of Moore&Atkeson and Peng&Williams on the maze task depicted in the top of Figure 2. The reward received at each time step is -1 and the discount factor is 0.99. The agent can take four actions, corresponding to the four compass directions, which stochastically move the agent to a different square. The bottom of Figure 2 shows the relative action outcomes of a 'north' action. In free space, an action can result in 15 possible successor states, each with equal probability. When the agent is close to a wall, this number decreases.

To obtain an upper bound on the performance, we also compared against a method that performs value iteration (until convergence) at each time step, using the most recent model estimate.

As exploration strategy, the agent select with 5% probability a random action, instead of the greedy one. On top of that, we use the 'optimism in the face of uncertainty' principle, as also used by Moore & Atkeson. This means that as long as a state-action pair has not been visited for at least M times, it's value is defined as
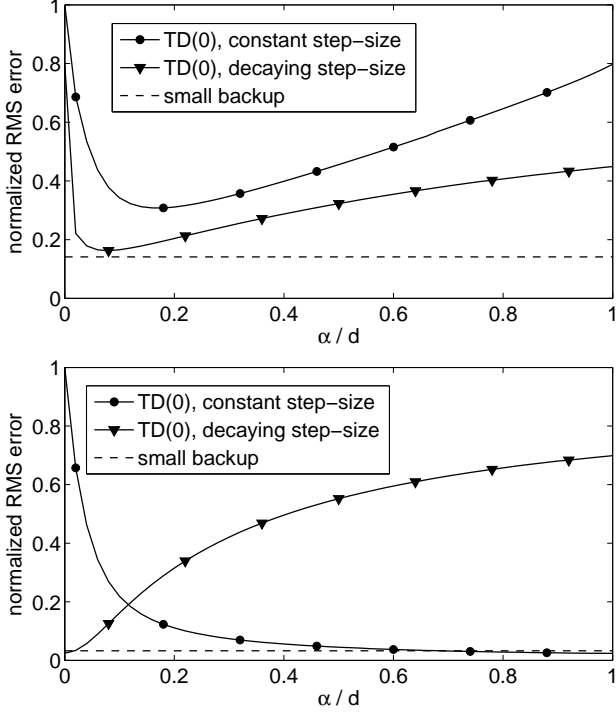
Figure 1. Average RMS error over the first 10.000 observations, normalized by the initial error, for different values of the step-size parameter α, in case of constant step-size, or different values of the decay parameter d, in case of decaying step-size. The top graph corresponds with the first evaluation task; the bottom graph with the second.

some optimistically defined value (0 for our maze task), instead of the value based on the model estimate. We optimized M for the value iteration method, resulting in $M = 4$, and used this value for all methods.

We performed experiments for 1, 3, 5 and 10 update cycles per time step. Figure 3 shows the average return over the first 200 episodes for the different methods. The results are averaged over 100 runs. The maximum standard deviation is 0.1 for all methods, except for the method of Peng & Williams, which had a maximum standard deviation of 1.0.

The computation time per update cycle was about the same for the three different PS implementations, with a small advantage for the small backup implementation, which shows that the $\mathcal{O}(P_{re})$ computation (see Table 1) is dominant in this task. The computation time per observation of the value iteration method was more than 400 times as high as a single update cycle.

PS with small backups turns out to be very effective. With only a single update cycle, the value-iteration result can be closely approximated, in contrast to the

two classical implementations. The results also show that the Peng & Williams method performs considerably worse than the one of Moore & Atkeson in the considered domain. This can be explained by the different backups they perform. The effect of the backup of Peng & Williams is proportional to the transition probability, which in most cases is $\frac{1}{15}$. In contrast, the Moore & Atkeson method performs a full backup each update cycle. While the small backup implementation also uses backups that are proportional to the transition probability, it performs a lot more backups per update cycle. Specifically, a number that is proportional to the number of predecessors. In general, this number will increase when the stochasticity of the domain increases.
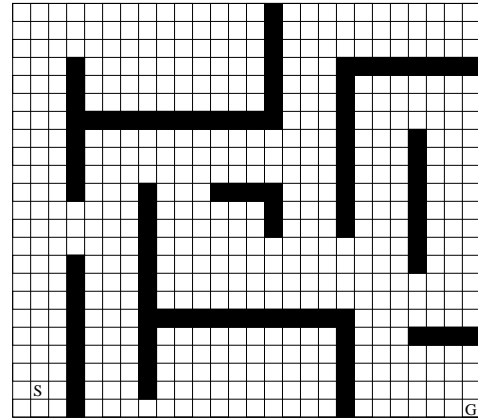

Figure 2. Above, the maze task, in which the agent must travel from S tothe G. Below, transition probabilities $(\cdot\frac{1}{15})$ of a 'north' action for different positions of the agent (indicated by the circle) with respect to the walls (black squares).

6. Discussion

Prioritized sweeping can be viewed as a generalization of the idea of replaying of experience in backward order (Lin, 1992), which by itself is related to *eligibility traces* (Sutton, 1988; Watkins, 1989; Sutton & Singh, 1994). What all these techniques have in common is that new information (which can be value changes, but at its core all value changes originate from new data) is propagated backwards. Whereas backward replay and eligibility traces use the recent trajectory for backward
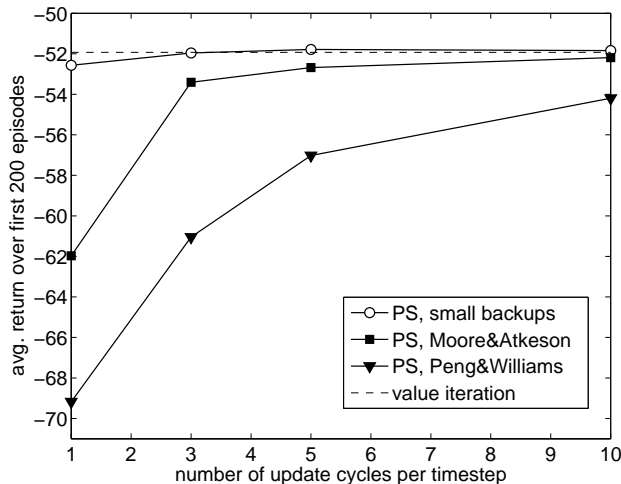
Figure 3. Performance of the PS implementations on the maze task for a different number of update cycles per time step and a method that performs value iteration at each time step.

propagation of information, prioritized sweeping uses a model estimate for this. Hence, it propagates new information more broadly.

What gives the performance edge to the small backup implementation is that it implements the principle of backward updating in a cleaner and more efficient way. One update cycle of Algorithm 2 represents, in a way, the ultimate backwards backup: all predecessors are updated with the current value of a chosen state, which is selected because it recently experienced a large value change. In contrast, the other PS implementation place the predecessors in a queue and backup only the state with the highest priority in the next update cycle. On top of that, the computation time per update cycle is lower for the small backup implementation (see Table 1).

The new implementation of PS introduced in this paper would be impossible without the new backup. The small backup allows for very targeted updates that are computationally very cheap. This enables finer control over how computation time is spend, which is what drives the new PS implementation.

7. Conclusion

We demonstrated in this paper that the planning step in model-based reinforcement learning method can be done substantially more efficient by making use of small backups. These backups are finer-grained version of a full backup, which allow for more control over how the available computation time is spend. This

makes new, more efficient, update strategies possible. In addition, small backups can be useful in domains with very tight time constraints, offering a parameter-free alternative to sample backups, which were up to now often the only feasible option for such domains.

References

Kaelbling, L. P., Littman, M. L., Moore, A. P. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285.

Lin, L.J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3):293–321.

Moore, A., Atkeson, C. (1993). Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130.

Peng, J., Williams, R. J. (1993). Efficient learning and planning within the dyna framework. *Adaptive Behavior*, 1(4):437–454.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine learning*, 3(1): 9–44.

Sutton, R. S., Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, Massachussets.

Sutton, R. S., Singh, S. P. (1994). On step-size and bias in temporal-difference learning. In *Proceedings of the Eight Yale Workshop on Adaptive and Learning Systems*.

Watkins, C. (1989). *Learning from delayed rewards.* PhD thesis, King's College, Cambridge, England.